Run Control Parameter (RCP) Objects in DØ

Harrison B. Prosper
Marc Paterno

November 11, 1998

1 Introduction

This informal note is neither a design nor a requirements document and should not be construed as either! Nor is it an exhaustive "core dump" of DØs understanding of, and use of, RCP objects. Rather I offer it as a sketch to orient further detailed discussion. The currently active members of the DØ RCP group are: Jim Kowalkowski, Marc Paterno and myself. Marc is the principal architect of the new RCP system and Jim has built a more sophisticated version of the parser, which we plan to install shortly. My role is largely supervisory, although I plan to get my hands dirty soon! My hope, of course, is that Vicky Whites "group" will supply, in part, the expertise needed to address the database issues.

In Run I, DØ used Run Control Parameter (RCP) objects to control and configure executable programs; for example, DØRECO, the DØ reconstruction program. We plan to use RCP objects, similarly, in Run II. At an abstract level, an RCP object is a container of (**name**, **value**) pairs, which we refer to as parameters. The **name** is of type **string**. It identifies the parameter. The **value**, of the parameter, can be one of the following types

- bool
- int
- float
- string (from the C++ Standard Library)

- RCP

or vectors (again, from the C++ Standard Library) thereof. Note that RCP objects can contain RCP objects. This is a simple way to support (in principle, arbitrarily complicated) structures—a requirement imposed upon us by users. Such nesting can lead (presumably unintentionally) to a circular RCP object. The RCP parser should reject any script that would lead to such a structure.

2 Properties of RCP objects

An RCP object must

- be capable of being made persistent;
- be read-only;
- have unique identity determined by content—two RCP objects are considered to be the same object if and only if their contents are the same;
- contain a single instance of a (small) object, called RCPID, that serves as a proxy for the identity of the RCP object—if the RCPIDs of two RCP objects are equal if and only if their RCP objects are equal;
- impose the requirement that each parameter, within the scope of the RCP object, have a unique name and
- require the collection of parameters to form a set (as opposed to a multiset).

The RCPID object encapsulates the identity and origin of an RCP object. We need to know the origin of RCP objects to ensure that critical programs, like the DØ reconstruction program, will only use RCP objects that have been released via the DØ CVS code repository. There was no way in the Run I system to enforce the use of properly released RCP objects; it was possible to run the reconstruction program with a rogue RCP object that had not been officially released. (Of course, this was generally done for reasons judged good, but alas not good enough to be documented!) In Run II we plan to

do better; we must be able to tell whether an RCP object is official or not. The RCPID is a computer-friendly bookkeeping mechanism. It is not meant to be the agent through which a user (normally) will ask for an RCP object. Instead, one would ask for an RCP object by supplying to the RCP manager (another object) a meaningful triplet of strings: (**package**, **name**, **version**).

- **package** – This is the name of a DØ CVS package—a collection of physically and logically related code. Since it is certain that some RCP objects will be needed that are not related to any code released into CVS, the RCP system's set of packages will be a superset of the packages in the CVS repository.

- **name** – This is a meaningful name assigned to an RCP object. (The search protocol for finding an RCP object is first to search the local file system, then a group area and finally the RCP Database. When searching the local file system the RCP manager would be looking for an RCP script [see below] of the specified name, along a well defined relative path.)

- **version** – This is the name of a CVS code release version number. This is of relevance only to those RCP objects which are introduced into the database through CVS. Each RCP manager is configured to make requests for RCP objects from one and only one CVS release.

Since the same RCP object could be associated with many CVS code releases there will be, in general, a one-to-many map between RCPIDs and triplets. Each RCP object has methods to query its contents:

- bool **getBool**(string name) const;
- vector<bool> **getVBool**(string name) const;
- int **getInt**(string name) const;
- vector<int> **getVInt**(string name) const;
- float **getFloat**(string name) const;
- vector<float> **getVFloat**(string name) const;

- string **getString**(string name) const;
- vector<string> **getVString**(string name) const;
- RCP **getRCP**(string name) const;
- vector<RCP> **getVRCP**(string name) const;
- RCPID **getRCPID**()

There is also a method to print the RCP object.

3 RCP Scripts

The specification for constructing an RCP object can be provided in an (ASCII) RCP script, sometimes called an RCP file. Here is an example of an RCP script (from Jim Kowalkowski)

```
# test file
int y = 3
int c = 456     # another comment

// comment style
float z0 = 4.5
float z1 = 4.   // comment style
float z2 = .5

string a = this is a test
string b = this \silly\\thing\ works!
string d = 56his string e = 5 string f = word

int y=5
int h = ( 1,2,3,4,5,6 )
string i = ( this,is,a,test)

RCP scum0 = (SUSY, set2)                # package = SUSY, name = set2
RCP scum1 = { int t= 4, float y  }    # inline RCP object
RCP scum2 = { int t= 4, float y  }
bool YESNO = true
```

This example contains all the basic scalar types and as well as examples of vectors. Here are a few points to note:

- The language is simple and is strongly typed.
- The order of parameters in a script is unimportant.
- An embedded RCP object (e.g., scum0) is specified using a pair, (name, value). The version is not listed, and will be that associated with the RCPManager object used to create the RCP object.
- The elements of a vector are of the same type.

Jim Kowalkowski has created a prototype of a parser, using lex and yacc, that is able to parse such scripts.

4 The RCP Database (RCPDb)

Our need to enforce uniqueness of RCP objects, and to make them persistent over many years, makes a strong case for the use of a real database. However, it is not necessary to use an object-oriented database to store objects; a relational database will do. Currently Oracle is the database of choice. The RCP (Oracle) database (RCPDb) is the ultimate repository of RCP objects. The CVS repository is the ultimate repository of official RCP scripts. These scripts serve as the specification, blueprint if you will, for those RCP objects destined to be used in critical DØ programs, like DØRECO. Other RCP objects can enter RCPDb via specifications that can come via the Web. We anticipate that a large fraction of the collaborations Monte Carlo event generation will be done at sites remote from Fermilab. We failed miserably in Run I to provide the necessary tight (and automatic) binding between Monte Carlo files and the specifications for their creation. We plan to effect this binding by embedding RCPIDs in Monte Carlo events. For this really to work as we wish, a very high level of reliability is required in the functioning of the database and the interfaces to it. RCP objects are to be stored in RCPDb in a fully dissected form. This will allow us to use standard SQL tools to query, in detail, RCP objects. There will be write-only access from the CVS repository to RCPDb, the updating of which will occur automatically upon each code release from CVS. Once updated, there will be an automatic

extraction of RCPDb to a read-only, exportable, file system for the benefit of sites remote from Fermilab. The RCP objects in the extracted database will, most likely, be stored in a manner that renders access to them as efficient as possible. A human-readable form of any RCP object can be had by invoking the objects print method.

4.1 A No-brainer Dissection of RCP Objects Into Tables

RCP objects are not mysterious things—mathematically they are trees: the RCP object is the trunk, which has branches (the parameters); each branch has one or more twigs (that is, a name, a type, plus one or more values) and sometimes a branch may be the trunk of an entire sub-tree (an embedded RCP object). Therefore, it is straightforward to map an RCP object to a tree of tables. I give an example below. It is not clever.

4.1.1 Example

An RCP object could be dissected into the following tables. Here Im using the python list syntax to represent the list of tables and a python tuple to represent a table. The syntax and semantics within the tuple are essentially those of MSQL.

```
Tables = [(RCP,       # Table Name
           package     char(32) not null,  + \
           name        char(32) not null,  + \
           comment     char(160),  + \
           thedate     date,  + \
           username    char(32),  + \
           rcpid       int not null, + \
           origin      int not null),

(RCP_VERSION,
version        char(16) not null,  + \
rcpid  int not null),

(RCP_PARAMETER,
```

```
name   char(32) not null,  + \
type   char(8) not null,  + \
comment     char(80),  + \
typeid         int not null,  + \
rcpid  int not null),

(RCP_INT,
value  int not null,  + \
typeid         int not null),

(RCP_FLOAT,
value  real not null,  + \
typeid         int not null),

(RCP_BOOL,
value  int not null,  + \
typeid         int not null),

(RCP_STRING,
value  char(32) not null,  + \
typeid         int not null),

(RCP_RCP,
rcpid  int not null,  + \
typeid         int not null)]
```

The **rcpid** and **typeid** provide links across tables. As noted above, the RCPID object encapsulates the fields **rcpid** and **origin**.

It is likely that this obvious structure is not optimal. The cleverness comes in understanding how, in practice, to do the mapping between RCP objects and tables so that the system is

1. easy to understand, and therefore easy to maintain over many years;
2. not brittle (it can be extended painlessly if new types are introduced);
3. allows for efficient insertion and extraction of RCP objects;

4. allows for the efficient testing for the equality of RCP objects;

5. allows for efficient queries of RCP contents (*e.g.*, which set of RCP objects contains ConeSize = 0.45 and Jet1EtCut = 20 GeV?), and can cope with, say, 50 users at once.

5 Summary

Our aim in building, from scratch, a new RCP system for Run II is to effect a marked improvement in the bookkeeping of data-sets be they from the Tevatron, or from a remote Monte Carlo farm. We wish to build a system that will provide a tight and automatic binding between the instructions for creating the data-sets and the data-sets. The RCP mechanism is our proposed solution. Five years from now I should like to be able to find out precisely how a given SUSY/Higgs file was generated. More importantly, I must be able to say, with a high degree of confidence how my ten golden Higgs candidates were reconstructed!